Chapitre 1 : Introduction à l'algorithmique

1.1 Définitions

Algorithme :

Un algorithme est une suite d'instructions. L'exécution correcte de ces instructions dans un ordre bien défini conduit à la résolution d'un problème.
Un ordinateur n'est fondamentalement capable de comprendre que quatre catégories d'instructions :

- L'affectation des variables
- La lecture et l'écriture
- Les tests
- Les boucles

Pour représenter les algorithmes on utilise plusieurs types de notations, en particulier :

- Les organigrammes (représentation graphique)
- Le pseudo-code

Exemple :
Algorithme qui teste si un nombre saisi au clavier est négatif, positif ou nul.

<u>Pseudo-code</u> :

#partie déclaration
Variable N en Entier

#corps de l'algorithme
Début
Ecrire ("Donner un nombre entier")
Lire N
Si N < 0 alors
    Ecrire ("Ce nombre est négatif ")
Sinon
    Si N > 0 alors
    Ecrire ("Ce nombre est positif")
    Sinon
    Ecrire ("Ce nombre est nul")
    Finsi
Finsi
Fin

Organigramme :

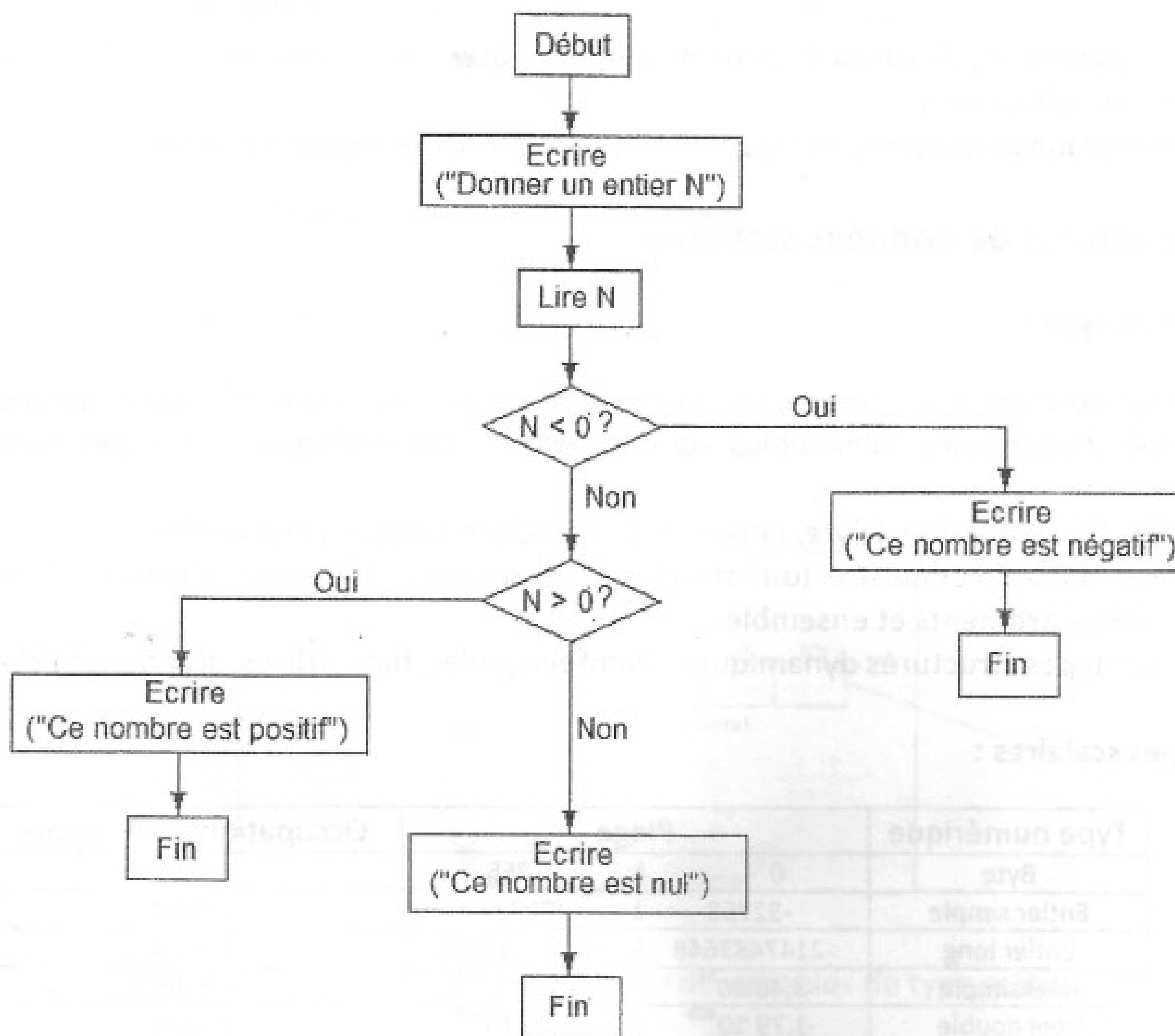


Figure 1.1 Exemple d'organigramme

Un programme est un algorithme avec des structures de données écrit dans un langage de programmation (Pascal, C, Fortran, C++, Java, Python,..., etc).

Un programme pour être exécuté sur un ordinateur doit être traduit (interprété ou compilé) dans le langage de la machine. Cette traduction produit un programme qui porte le nom d'exécutable.

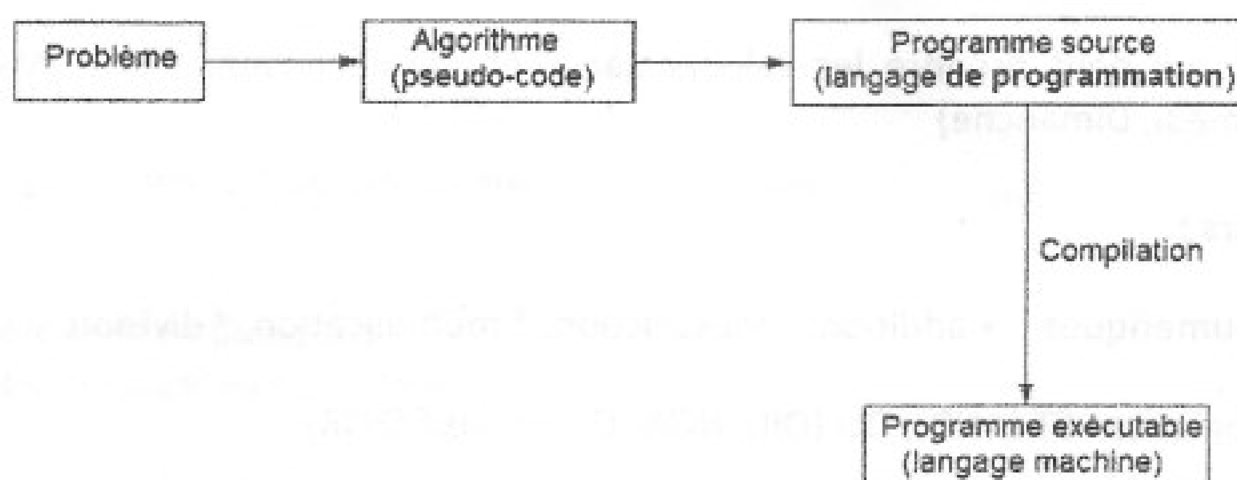


Figure 1.2 Processus de résolution d'un problème par ordinateur

Historique :

Euclide : algorithme de calcul du pgdc de deux nombres entiers
Archimède : calcul de π
Mohammed Ibn Al Khawarizmi : représentation décimale et règles de calcul

1.2 Structures de données statiques

Notion de type :

Le typage consiste à associer à un objet (variable) un ensemble de valeurs possibles et un ensemble d'opérations admissibles sur ces valeurs. On distingue différentes catégories de types :

- les types scalaires : Byte, Entier, Réel, Caractère, Booléen et énuméré.
- Les types composés (ou structurés) statiques : Tableaux, chaînes de caractères, enregistrements et ensembles.
- Les types structurés dynamiques : Pointeurs, piles, files, arbres, graphes et fichiers.
-

Les types scalaires :

Type numérique	Plage			Occupation en mémoire
Byte	0	à	255	1 octet
Entier simple	-32768	à	+32767	2 octet
Entier long	-2147483648	à	+2147483648	4 octet
Réel simple	$-3,40\ 10^{38}$	à	$+3,40\ 10^{38}$	4 octet
Réel double	$-1,79\ 10^{308}$	à	$+1,79\ 10^{308}$	8 octet

Table 1.1 Plages et occupations mémoires des types numériques
dans un microordinateur

Type caractère (alphanumérique) : code ASCII (1 octet) ou Unicode (2 octets)

Type booléen : ne peut prendre que la valeur logique VRAI ou FAUX

Type énuméré : c'est un type défini par l'utilisateur.

Exemple :
Le type jour qui peut prendre les valeurs dans l'ensemble {Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche}

Les opérateurs :

Opérateurs numériques : + addition, - soustraction, * multiplication, / division

Opérateurs logiques : ET (AND), OU (OR), NON, OU exclusif (XOR)

Opérateurs de comparaison : = égal, <> différent, < inférieur, > supérieur, =< inférieur ou égal, >= supérieur ou égal.

Déclaration de variables :

Une variable est une zone mémoire repérée par une étiquette. La taille de la variable dépend du type dont lequel elle a été déclarée.

Exemple : *Variable X en Entier*

Variables Y, Z en Réel

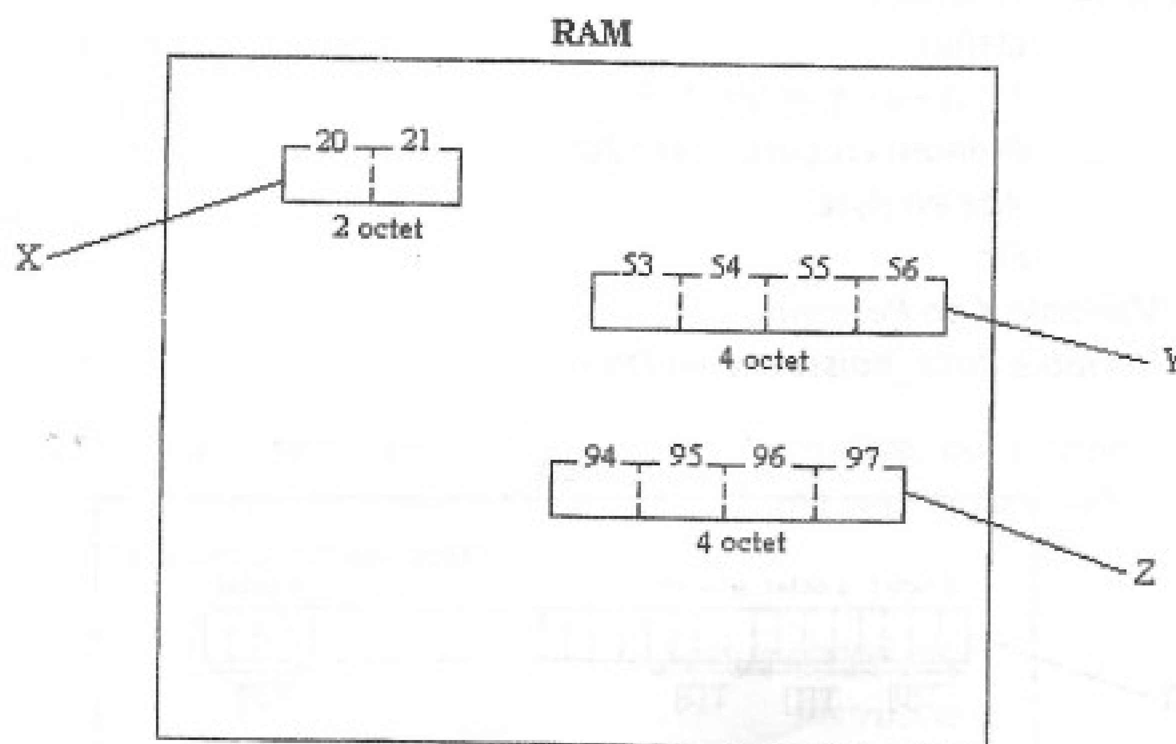


Figure 1.3 Représentation en mémoire de variables de type scalaire

Les types composés statiques :

Les tableaux : un tableau est une structure constituée d'un regroupement de données de même type. Les données individuelles sont repérées par un indice. L'indice est de type entier. Les opérations possibles sur les tableaux se limitent à la manipulation des composantes individuelles.

Exemple :

Tableau T[10] en Réel
Tableau Age[50] en Byte
Tableau M[20,20] en Réel

Les chaînes : une chaîne est un groupement de caractères

Exemple :

Variable nom en Caractères
*Variable prénom en Caractère*25*

Les enregistrements (ou structures) : un enregistrement est constitué d'un regroupement de données qui peuvent être hétérogènes (de différents types).

Exemple :

Structure Date
Début
Jour en Entier
Mois en Entier
Année en Entier
Fin

Structure Personne
Début
*Nom en Caractère*25*
*Prénom en Caractère*20*
Age en Byte
Fin
Variable X en Personne
Variable date_naissance en Date

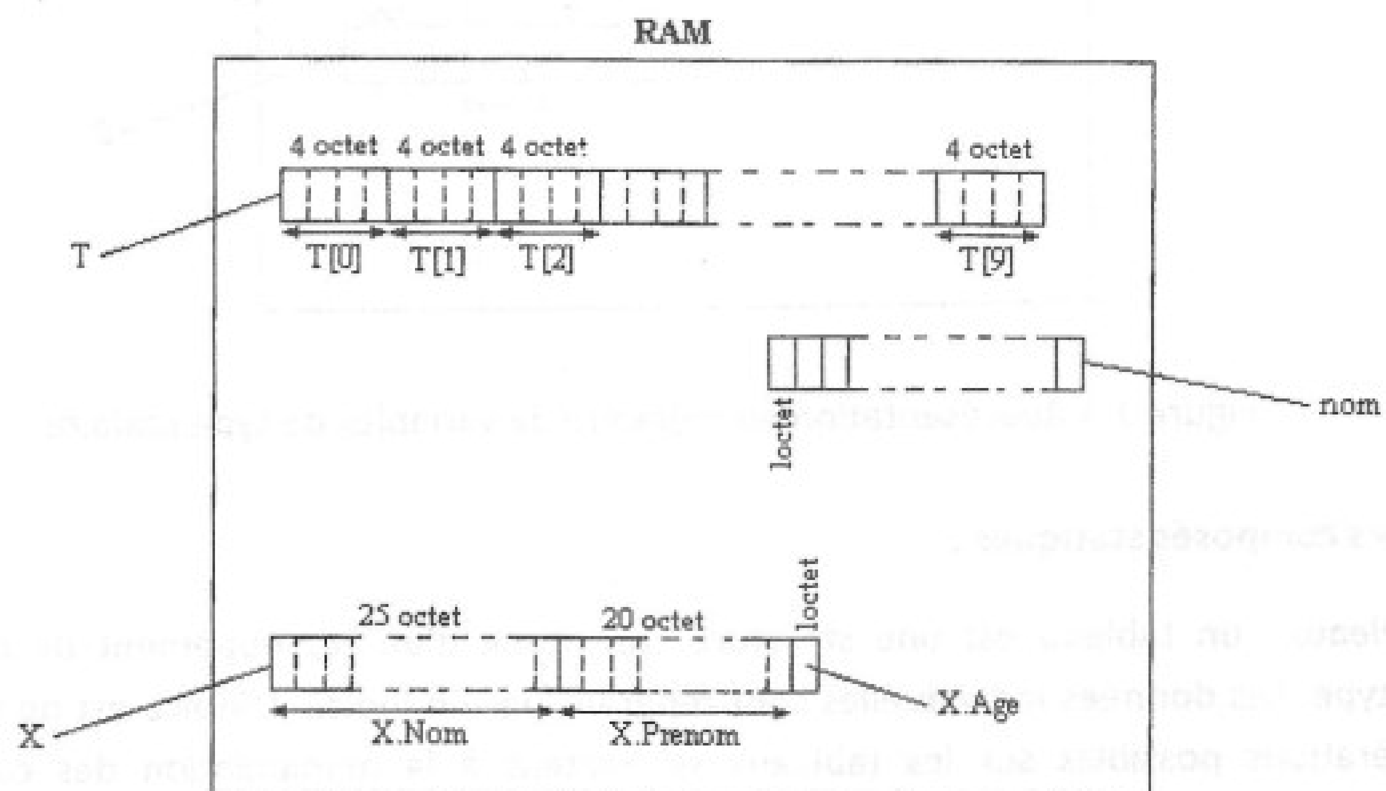


Figure 1.4 Représentation en mémoire de variables de type composé

1.3 Les instructions de base

L'affectation :

L'affectation est l'opération qui consiste à attribuer une valeur à une variable, c.a.d : remplir la zone mémoire qui porte le nom de la variable par cette valeur.

Exemple : *Variable X en Entier*
Variable Y en Caractères
Début
X <----- 12
Y <-----"Faculté des Sciences"
Fin

Lecture /Ecriture :

La lecture c'est l'opération de saisie des données (en général du clavier)
L'écriture c'est l'opération d'affichage des données (en général sur écran)

Exemple :
***Lire** X*
***Lire** X, Y*
Ecrire (**" Filière SMP "**)
Ecrire (**" Le nombre d'étudiants est ", X **)
La chaîne entre guillemets est affichée telle qu'elle est écrite dans l'instruction *Ecrire*, tandis que X sera remplacée par sa valeur.

Les tests :

C'est une structure logique, appelée aussi structure alternative, qui permet l'exécution d'une ou plusieurs instructions selon la valeur VRAI ou FAUX d'une expression logique.
Il y a deux formes possibles pour un test :

Si** expression **alors
Instructions
Finsi

Si** expression **alors
Instructions
Sinon
Instructions
Finsi

Exemple :
Variable T en Entier
Début
Ecrire ("Donner la température de l'eau")
Lire T
Si T =< 0 alors
Ecrire ("C'est de la glace")
Sinon
Si T < 100 alors
Ecrire ("C'est du liquide")
Sinon
Ecrire ("C'est de la vapeur")
Finsi
Finsi

Remarque :
D'après l'exemple précédent on remarque qu'on peut imbriquer les tests.

Dans le cas où l'expression logique des tests imbriqués est réduite au test de la valeur d'une variable on peut utiliser l'instruction suivante :

Cas où *Variable* ***vaut :***
Valeur1) *Instructions*
Valeur2) *Instructions*
Valeur3) *Instructions*
Autrement) *instructions*
Fincas

Exemple :

Lire X
Cas où X vaut :
1) *Ecrire ("Lundi")*
2) *Ecrire ("Mardi")*
3) *Ecrire ("Mercredi")*
4) *Ecrire ("Jeudi")*
5) *Ecrire ("Vendredi")*
6) *Ecrire ("Samedi")*
7) *Ecrire ("Dimanche")*

Autrement) Ecrire ("Ce numéro ne correspond à aucun jour de la semaine")
Fincas

Les boucles :

C'est une structure logique, appelée structure répétitive, qui permet l'exécution d'une ou plusieurs instructions d'une manière répétitive. Elle est contrôlée par la valeur VRAI ou FAUX d'une expression logique ou par un compteur.
Il existe trois types de boucles :

Tantque *expression* ***faire***
Début
Instructions
Fin

Exemple :

```
#calcul de S=1+2+3+4+...+N avec N entier positif donné
Variables N, I , Somme en Entier
Début
Ecrire ("Donner un entier positif")
Lire N
I <------- 1
Somme <-------- 0
Tantque ( I <= N ) faire
        Début
        Somme <-------- Somme + I
        I <------- I + 1
       Fin
Ecrire ("La somme est :", Somme)
Fin
```

Faire

Début

Instructions

Fin

Tantque *expression*

Exemple :

```
#calcul de S=1+2+3+4+...+N avec N entier positif donné
Variables N, I , Somme en Entier
Début
Ecrire ("Donner un entier positif")
Lire N
I <------- 1
Somme <-------- 0
faire
        Début
        Somme <-------- Somme + I
        I <------- I + 1
       Fin
Tantque ( I <= N )
Ecrire ("La somme est :", Somme)
Fin
```

Pour *Compteur <------ ValInitiale* ***à*** *Valfinale* ***Pas*** *ValPas* ***faire***

Début

Instructions

Fin

Si *ValPas* vaut 1 on peut l'omettre

Exemple :

```
#calcul de S=1+2+3+4+...+N avec N entier positif donné
Variables N, I , Somme en Entier
Début
Ecrire ("Donner un entier positif")
Lire N
Somme <-------- 0
Pour I <------- 1 à N faire
        Début
        Somme <-------- Somme + I
       Fin
Ecrire ("La somme est :", Somme)
Fin
```

Remarque :

On peut imbriquer des boucles.

Exemple :

```
#calcul de S=1!+2!+3!+4!+...+N! avec N entier positif donné
Variables N, P,  I ,  J, S en Entier
Début
Ecrire ("Donner un entier positif")
Lire N
S <-------- 0
Pour I <------- 1 à N faire
        Début
        J <-------- 1
        P <-------- 1
        Tantque  J =< I  faire
            Début
                P <-------- P*J
                J <-------- J+1
            Fin
        S<-------- S + P
       Fin
Ecrire ("La somme est :", S)
Fin
```

1.4 Les fonctions

Les fonctions prédéfinies :

Partie entière	*X <-------Ent(3,25)*	*X* vaut *3*
Modulo	*X <-------Mod(10,3)*	*X* vaut *1*
Alea	*X <-------Alea( )*	*0=< X < 1*
Sin, Cos, Log......		
Concaténation	Y <------- *"Bon" & "jour"*	*Y* vaut *"Bonjour"*

Les fonctions personnalisées :

Ce sont des fonctions écrites par l'utilisateur dans un programme et peuvent être appelées comme les fonctions prédéfinies. On distingue deux types de fonctions :

- Les fonctions avec une valeur de retour
- Les fonctions sans valeur de retour appelées parfois procédures

Exemples :

```
#fonction qui renvoie la moyenne des éléments d'un tableau
Fonction Moyenne (T Tableau en Réel, N en Entier positif) en Réel
Début
Variables S, M en Réel
S <-------- 0
Pour I <------- 0 à N-1 faire
        Début
        S<-------- S + T[I]
       Fin
```

```
M <-------- S/N
Renvoyer (M)
Fin

#fonction qui donne pile ou face
Fonction Test( )
Début
Variable T en Réel
T <-------- Alea( )
Si T < 0.5 alors
        Ecrire ("Pile")
Sinon
        Ecrire("Face")
Finsi
Fin

#fonction qui échange le contenu de deux variables réelles
Fonction Echange(A en Réel,  B en Réel )
Début
Variable X en Réel
X<-------- A
A<-------- B
B<-------- X
Fin
```

Fonctions récursives :

Une fonction récursive est une fonction qui, au cours de son exécution, appelle elle-même

Exemple :

On a la relation N !=N*(N-1) !

```
Fonction Fact(N en Entier positif ) en Entier
Début
Si N = 0 alors
        Renvoyer (1)
Sinon
        Renvoyer (N*Fact(N-1))
Finsi
Fin
```

Chapitre 2 : Algorithmes de recherche

2.1 Recherche séquentielle

Principe :

C'est l'algorithme de recherche le plus simple et le plus utilisé pour les tableaux de faible taille (jusqu'à 1000 éléments).

L'algorithme consiste à balayer le tableau des éléments du début vers la fin jusqu'à trouver l'élément recherché ou arriver à la fin du tableau.

Algorithme :

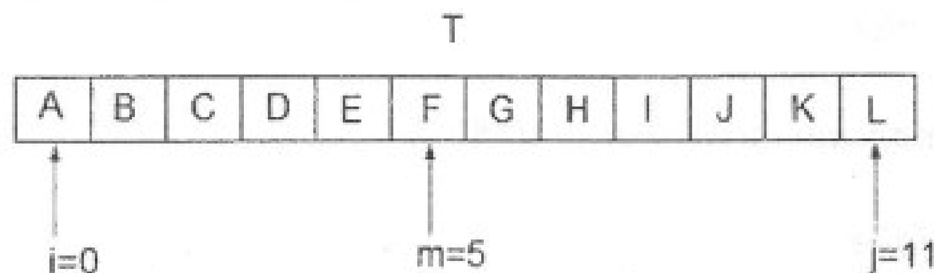
```
#Recherche séquentielle d'un élément dans un tableau d'entiers
Variables I, N, X  en Entier
Variable Trouvé en Booléen
Tableau T[1000] en Entier
Début
Ecrire ("Donner la dimension du tableau < 1000 ")
Lire N
Pour I <------- 0 à N-1 faire
        Début
        Lire T[I]
        Fin
Ecrire("Donner l'élément à chercher")
Lire X
I <-------- 0
Trouvé <-------- FAUX
Tantque (NON Trouvé) ET ( I < N) faire
          Début
              Si X=T[I] alors
                      Trouvé<--------VRAI
              Sinon
                      I<-------- I + 1
              Finsi
        Fin
Si Trouvé alors                    (ou si trouvé=vrai alors)
        Ecrire ("Elément trouvé à la position ", I)
Sinon
        Ecrire("Elément non trouvé")
Finsi
Fin
```

2.2 Recherche dichotomique

Principe :

C'est l'un des algorithmes de recherche les plus rapides et les plus utilisés dans le cas de tableaux de grande taille, mais qui ne s'applique qu'à des tableaux déjà triés.
Son principe consiste à réduire de moitié le nombre total des éléments à examiner à chaque itération.

Exemple :
Soit à chercher l'élément H dans le tableau suivant :



1ère Itération : on examine tout le tableau
Indice du milieu m=Ent((0+11)/2)=5
T[m]=F < H donc on doit chercher à droite, i=m+1=6.

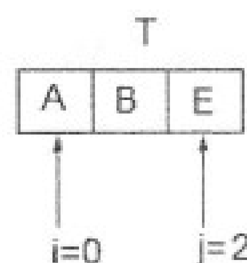
2ème Itération : on examine la partie du tableau de T[6] à T[11]
m=Ent((6+11)/2)=8
T[m]=I > H donc on doit chercher à gauche, j=m-1=7.

3ème Itération : on examine la partie du tableau de T[6] à T[7]
m=Ent((6+7)/2)=6
T[m]=G < H donc on doit chercher à droite, i=m+1=7.

4ème Itération : la partie du tableau à examiner est réduite à un seul élément T[7] qui est l'élément recherché m=7 et T[7]=H.

Et l'algorithme prend fin.
Si l'élément recherché ne figure pas dans le tableau il est nécessaire de faire un test supplémentaire jusqu'à ce que j devient inférieur à i.

Exemple :
Recherche de D dans le tableau suivant :



1ère Itération : on examine tout le tableau
m=Ent((0+2)/2)=1
T[m]=B < D donc on doit chercher à droite, i=m+1=2.

2ème Itération : on examine T[2]

m=Ent((2+2)/2)=2

T[m]=E > D donc on doit chercher à gauche, j=m-1=1.

Alors j<i , la recherche est non fructueuse et l'algorithme prend fin.
j<i est le test d'arrêt des itérations.

Algorithme :

```
#Recherche dichotomique d'un élément dans un tableau d'entiers
Variables I, J, N, M, X en Entier
Variable Trouvé en Booléen
Tableau T[1000] en Entier
Début
Ecrire ("Donner la dimension du tableau < 1000 ")
Lire N
Pour I <------- 0 à N-1 faire
        Début
        Lire T[I]
        Fin
Ecrire ("Donner l'élément à chercher")
Lire X
I <-------- 0
J <-------- N-1
Trouvé <-------- FAUX
M <-------- Ent((I+J)/2)
Tantque (NON Trouvé) ET ( J >= I ) faire
            Début
                Si X=T[M] alors
                        Trouvé<--------VRAI
                Sinon
                        Si T[M] > X alors
                                J<-------- M – 1
                        Sinon
                                I<-------- M +1
                        Finsi
                M <-------- Ent((I+J)/2)
                Finsi
        Fin
Si Trouvé alors
        Ecrire ("Elément trouvé à la position ", M)
Sinon
        Ecrire ("Elément non trouvé")
Finsi
Fin
```

Remarque :

Si le tableau est trié dans l'ordre décroissant, il suffit de changer l'inégalité T[M] > X par T[M] < X dans l'algorithme.

Chapitre 3 : Complexité des algorithmes

3.1 Définition de la complexité algorithmique

Temps d'exécution:

Pour résoudre un problème la question du choix de l'algorithme se pose souvent. En effet, dans la majorité des cas, l'algorithme choisi doit souvent satisfaire un compromis entre deux besoins qui sont souvent contradictoires :

- Simplicité à comprendre et facilité de mise en œuvre
- Exploitation optimale des ressources de la machine et plus précisément s'exécuter le plus rapidement possible

Ainsi, des algorithmes sophistiqués peuvent être non utilisables en pratique à cause de leur durée d'exécution énorme. On leur préfère des algorithmes simples moins efficaces mais qui ont des temps d'exécution raisonnables.

Le temps d'exécution d'un programme dépend des facteurs suivants :

- Les données entrantes dans le programme,
- La qualité du code généré par le compilateur pour la création du programme objet,
- La nature et la vitesse d'exécution des instructions du microprocesseur utilisé pour l'exécution du programme,
- La complexité algorithmique.

Notations O et Ω :

Le temps d'exécution d'un programme varie naturellement en fonction de la taille n des données traitées. Il s'exprime en fonction de n et est noté T(n).

Soient deux fonctions f et g positives définies sur $\mathbb{N}$.

On dit que T(n) est en O(f(n)) si :
$\exists\ k > 0$ et $n_0 > 0$ tel que $\forall\ n \geq n_0\ \ T(n) \leq kf(n)$

On dit que T(n) est en Ω (g(n)) si :
$\exists\ k > 0$ et $n_0 > 0$ tel que $\forall\ n \geq n_0\ \ T(n) \geq kg(n)$

Exemple :

$T(n) = 3n^3+2n^2$

$n_0 = 0$ et $k = 5$ $\quad \forall\ n \geq 0 \quad 3n^3+2n^2 \leq 5n^3$ $\quad$ T(n) est en $O(n^3)$
T(n) tend asymptotiquement vers n^3

$n_0 = 0$ et $k = 1$ $\quad \forall\ n \geq 0 \quad n^3 \leq 3n^3+2n^2$ $\quad$ T(n) est en $\Omega\ (n^3)$

Remarque :
Les fonctions f(n) les plus fréquemment rencontrées sont :
Logn, n, nlogn, n^2, n^3, n^4, 2^n, n!, n^n.

3.2 Exemples de calcul de complexité

Somme de n nombres:

Soit à calculer la somme S de n nombres $a_1, a_2, a_3, a_4,.., a_n$. L'algorithme utilisé se résume dans l'instruction suivante :

S <-------- $a_1+ a_2+ a_3+ a_4+...+ a_n$

On a n+1 accès en mémoire et n-1 addition. Ainsi, l'ordre de grandeur du temps d'exécution est T(n)=n+1+n-1=2n. Si on fait abstraction de la constante multiplicative 2, le temps d'exécution T(n) pour n grand est en O(n).

Calcul de e^x:

$S = e^x \approx 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + ... + \frac{x^n}{n!}$ avec *n* entier positif très grand.

```
#1er algorithme de calcul de la  valeur approchée de l'exponentielle
Variables i, j, n en Entier
Variables S, P,  x en Réel
Début
Ecrire ("Donnez l'ordre n ")
Lire n
Ecrire ("Donnez un réel ")
Lire x
S <-------- 1
i <-------- 1
Tantque (i=<M) faire                (1)
        Début
        P <-------- 1
        j<-------- 1
         Tantque (j=<i ) faire       (2)
           Début
          P <-------- P*x/j
          j <-------- j+1
          Fin
      S <-------- S+P
      i <-------- i+1
      Fin
Ecrire ("Valeur approchée de l'exponentielle de ", x, "est : ", S)
Fin
```

Le corps de la boucle intérieure (2) est exécuté 1+2+3+4+...+n $= \frac{n(n+1)}{2}$ fois. Les autres instructions de la boucle (1) sont exécutées n fois. Donc T(n) est en O(n²)

$n + \frac{n(n+1)}{2}$ ... à l'infini $\Rightarrow \frac{1}{2} n^2$

#2ème algorithme de calcul de la valeur approchée de l'exponentielle

```
Variables i, j, n en Entier
Variables S, P,  x en Réel
Début
Ecrire ("Donnez l'ordre n ")
Lire n
Ecrire ("Donnez un réel ")
Lire x
S <-------- 1
 i<-------- 1
P <-------- 1
Tantque (i=< n ) faire
        Début
            P <-------- P*x/i
            S <-------- S+P
            i<-------- i+1
        Fin
Ecrire ("Valeur approchée de l'exponentielle de ", x, "est : ", S)
Fin
```

Le corps de la boucle est exécuté n fois, donc T(n) est en O(n). On conclue que le deuxième algorithme est meilleur du point de vue temps d'exécution.

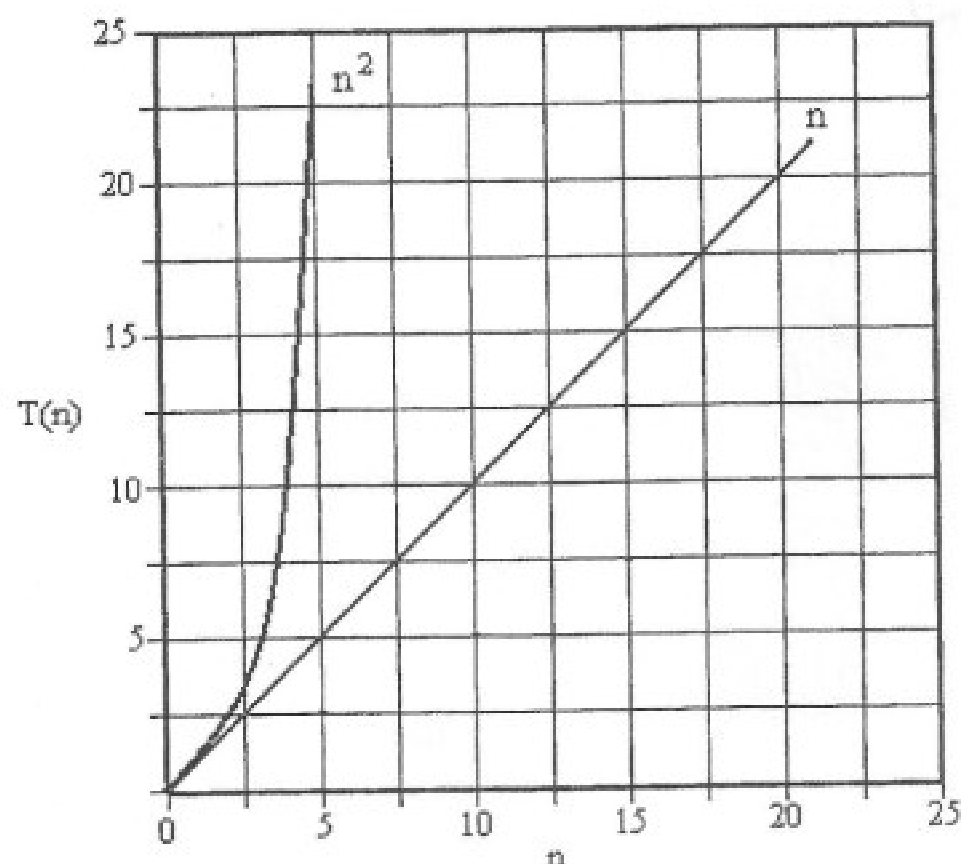


Figure 1.4 Temps d'exécution des deux algorithmes de calcul de l'exponentielle

Tri à bulles:

```
Fonction Tri_bulle(T Tableau en Réel, n en Entier)
Début
Variables i, j  en Entier
Pour i<------- 1 à n-1 faire      (1)
         Début
         Pour j <------- 0 à n-1-i  faire  (2)
                  Début
                  Si T[j]  > T[j+1] alors
                  Echange(T[j], T[j+1])   (3)
                   Finsi
                  Fin
        Fin
Fin
```

Dans le pire des cas si l'échange (3) est exécuté pour chaque j alors au total il sera exécuté n-i fois. Ainsi, il sera exécuté dans la boucle (1) $\sum_{i=1}^{n-1}(n-i)$ fois, c.à.d : l'instruction (3) est exécutée $\frac{n(n-1)}{2}$ fois . Dans ce cas le temps d'exécution T(n) est en O(n^2) .

Donner le temps d'execution dans :
le mauvais cas et le bon cas
(lorsque le tableau est donnée trié)
⇓
T(n) est en O(n²)

Chapitre 4 : Algorithmes de tri

4.1 Introduction

Un algorithme de tri est un algorithme qui permet d'organiser une collection d'objets selon un ordre déterminé. Les objets à trier font donc partie d'un ensemble muni d'une relation d'ordre. Les ordres les plus utilisés sont l'ordre numérique et lexicographique (dictionnaire).

Les algorithmes de tri sont classés selon deux caractéristiques principales importantes :

- La complexité algorithmique.
- La stabilité (conservation de l'ordre relatif des quantités égales pour la relation d'ordre)

Cette classification est très importante, car elle permet de choisir le type d'algorithme le plus adapté au problème traité, tout en tenant compte des contraintes imposées par celui-ci.

Les langages spécialisés en gestion des bases de données offrent des routines pré-établies pour les tris. Les langages non spécialisés (C, C++, Java,..., etc) permettent de développer des routines personnalisées équivalentes ou fournissent des algorithmes implémentés comme fonctions dans des bibliothèques.

Exemples d'algorithmes de tri :
Tri à bulle, algorithme quadratique dans le pire des cas (T(n) en $O(n^2)$) et stable
Tri par sélection, algorithme quadratique dans le pire des cas
Tri par insertion, algorithme quadratique dans le pire des cas et stable
Tri par comptage, algorithme linéaire ((T(n) en O(n)) et stable
Tri rapide (Quick sort)
Tri par fusion
Tri par tas
Tri par base

4.2 Tri à bulles

Principe :

Il consiste à parcourir le tableau à trier en effectuant la comparaison de deux éléments consécutifs et les échanger s'ils ne sont pas dans l'ordre.

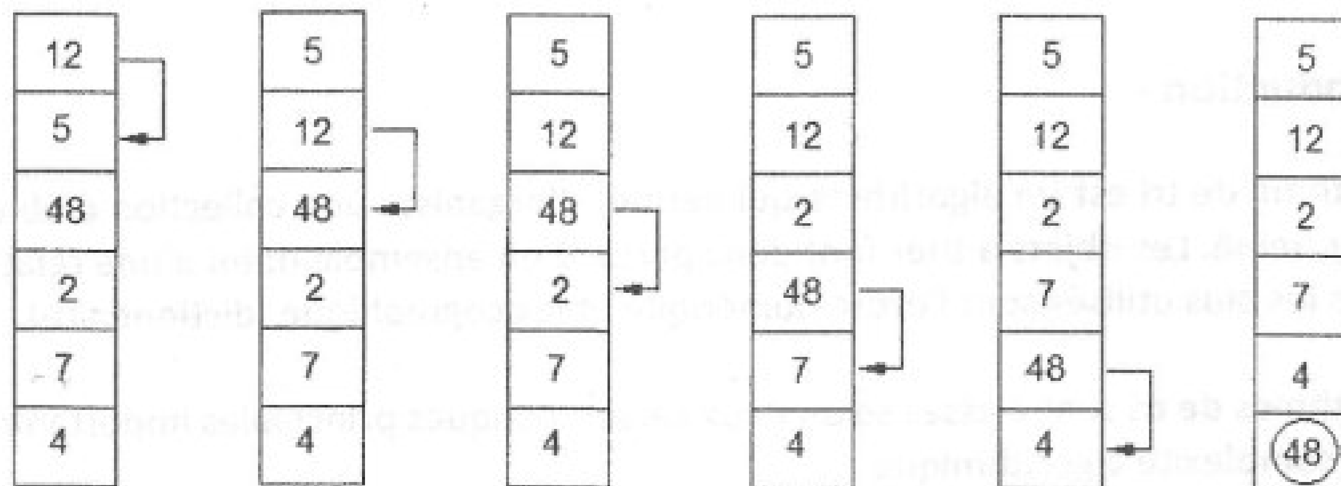
Cette opération a pour effet de déplacer, comme une bulle, l'élément le plus grand vers la fin du tableau.

Exemple :

Soit à trier le tableau suivant dans l'ordre croissant :

12	5	48	2	7	4

1ère itération



Après cette opération, l'élément le plus grand (48) est à sa place définitive.

Pour trier tout le tableau, il suffit de répéter la même opération N-1 fois (N nombre total des éléments du tableau)

Algorithme :

Fonction Tri_bulle(T Tableau en Réel, N en Entier)
Début
Variables I, J en Entier
Pour I <------- 1 à N-1 faire
    Début
    Pour J <------- 0 à N-1-I faire
        Début
        Si T[J] > T[J+1] alors
            Echange(T[J],T[J+1])
        Finsi
        Fin
    Fin
Fin

Remarque :
Il existe des améliorations du tri à bulle, on peut citer le tri Shaker et le tri Shell.

4.3 Tri par sélection

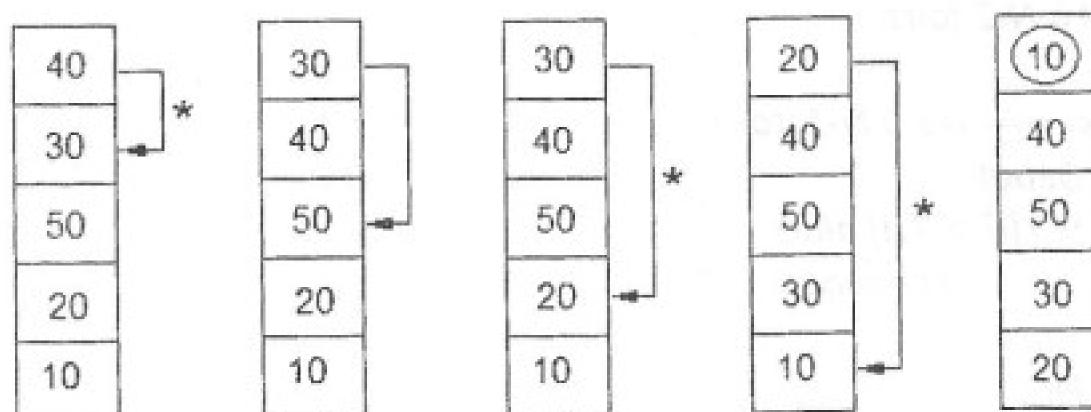
Principe :

Dans cet algorithme chaque itération consiste à extraire (ou sélectionner) un élément et le placer à sa position définitive. Ainsi, la 1ère itération extrait l'élément le plus petit et le place à la première position. La deuxième itération place l'élément suivant à la deuxième position et ainsi de suite.

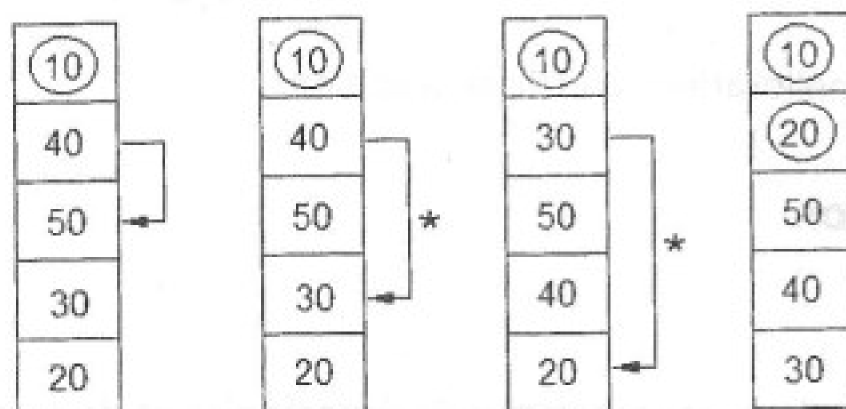
Exemple :
Soit à trier le tableau suivant dans l'ordre croissant :

40	30	50	20	10

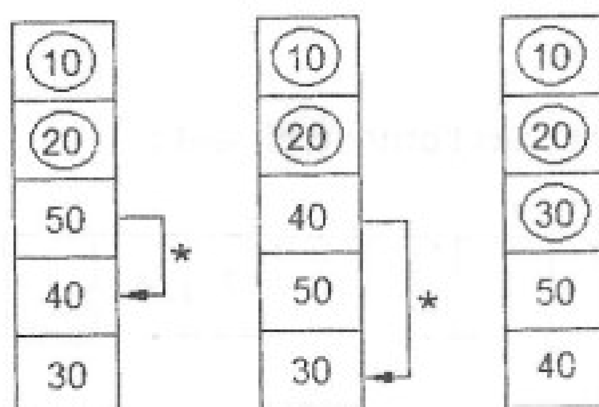
1ère itération



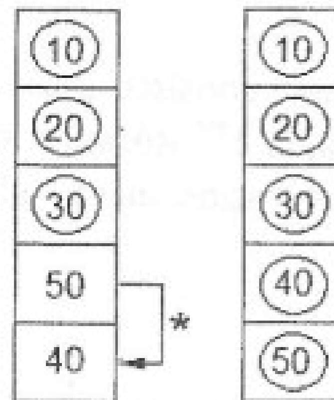
2ème itération



3ème itération



4ème itération



Algorithme :

```
Fonction Tri_sélection(T Tableau en Réel, N en Entier)
Début
~~Variable X en Réel~~
Variables I, J  en Entier
Pour I <------- 0 à N-2 faire
        Début
        Pour J <------- I+1 à N-1  faire
                Début
                Si T[I]  > T[J] alors
                        Echange(T[I],T[J])
                 Finsi
                Fin
      Fin
Fin
```

Remarque :

Il existe aussi des améliorations du tri par sélection.

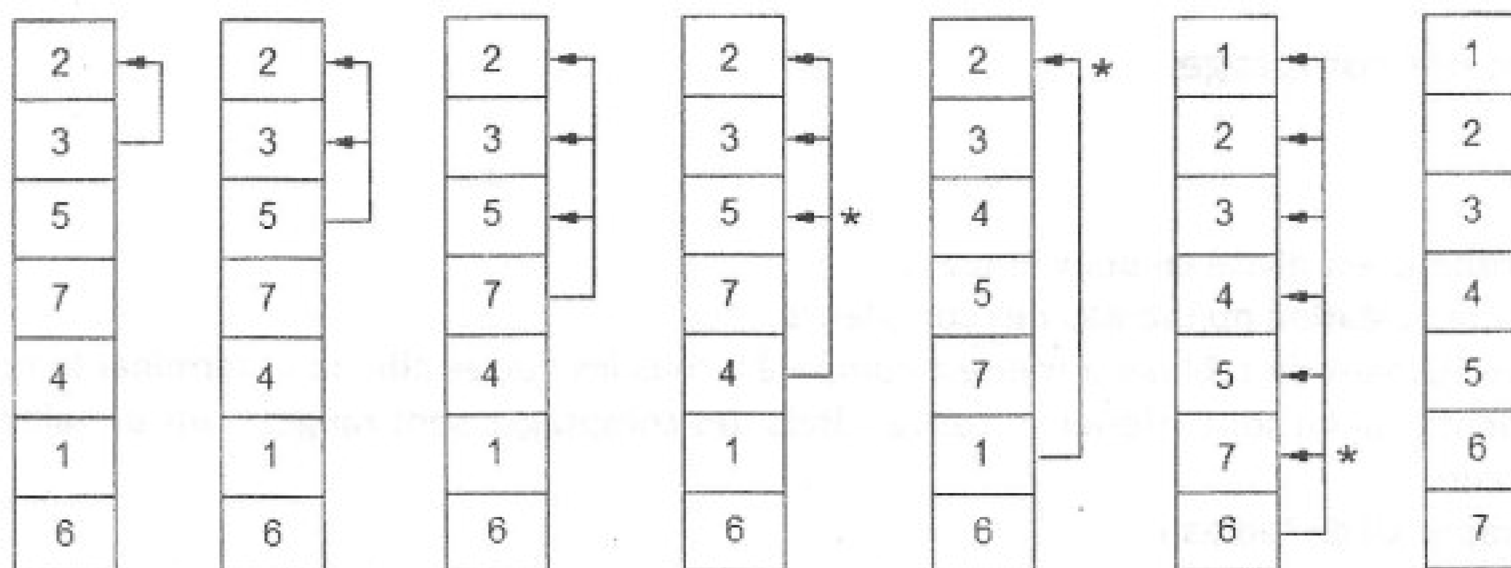
4.4 Tri par insertion

Principe :

L'algorithme consiste à parcourir le tableau dans l'ordre et de comparer chacun des éléments avec les éléments qui le précèdent jusqu'à trouver la place adéquate. Il ne reste plus qu'à décaler les éléments du tableau pour insérer l'élément considéré à sa place dans la partie déjà triée.

Exemple :
Soit à trier le tableau suivant dans l'ordre croissant :

2	3	5	7	4	1	6



Algorithme :

Fonction Tri_insertion (T Tableau en Réel, N en Entier)
Début
Variable X en Réel
Variables I, J, P en Entier
Pour I <------- 1 à N-1 faire
    Début
    P <------- 0
    Tantque (T[P] < T[I]) ET (P < I) faire
        Début
        P <------- P+1
        Fin
    #Sauvegarde de T[I]
    X <------- T[I]
    J <------- I -1
    #Décalage
    Tantque (J >= P) faire
        Début
        T[J+1] <------- T[J]
        J <------- J-1
        Fin
   #insertion
   T[P] <-------X
   Fin
Fin

4.5 Tri par comptage

Principe :

L'algorithme est divisé en deux étapes :
1ère étape : création du tableau des compteurs
Chaque élément du tableau à trier est comparé à tous les autres afin de déterminer le nombre d'éléments qui lui sont inférieurs. Les résultats des comptages sont rangés dans un tableau de compteurs.
2ème étape : tri du tableau
Le premier élément du tableau non trié est permuté avec celui dont le compteur est nul. Les éléments correspondants dans le tableau de compteurs sont aussi permutés. Le processus est appliqué une deuxième fois avec l'élément dont le compteur est égal à 1. Ceci est répété jusqu'à une valeur du compteur égal à la dimension du tableau diminuée de 1

Exemple :
Soit à trier le tableau suivant dans l'ordre croissant :

50	70	10	20	40	30

T	50	70	10	20	40	30
C	4	5	0	1	3	2

T	(10)	70	50	20	40	30
C	0	5	4	1	3	2

T	(10)	(20)	50	70	40	30
C	0	1	4	5	3	2

T	(10)	(20)	(30)	70	40	50
C	0	1	2	5	3	4

T	(10)	(20)	(30)	(40)	70	50
C	0	1	2	3	5	4

T	(10)	(20)	(30)	(40)	(50)	(70)
C	0	1	2	3	4	5

Algorithme :

```
Fonction Tri_comptage (T Tableau en Réel, N en Entier)
Début
Tableau C[ ] en Réel
Variables I, J en Entier
#Comptage
Pour I <------- 0 à N-1 faire
        Début
        C[I] <------- 0
        Fin
Pour I <------- 0 à N-2 faire
        Début
        Pour J <------- I+1 à N-1 faire
        Début
                Si T[J] < T[I] alors
                C[I] <------- C[I]+1
                Sinon
                C[J] <------- C[J]+1
                Finsi
        Fin
        Fin
#Tri
I <------- 0
J <------- 0
Tantque (I < N)  faire
        Début
        Si C[J] <> I alors
                Tantque (C[J] <> I)  faire
                Début
                J <------- J+1
                Fin
        Echange(T[I], T[J])
        Echange(C[I], C[J])
        Finsi
        I <------- I+1
        J <------- I
        Fin
Fin
```

Remarque :

Cet algorithme ne convient que pour des tableaux qui ne contiennent pas des éléments dupliqués

4.6 Tri rapide

Principe :

C'est l'un des algorithmes les plus efficaces mis au point à l'origine par C.A.R HOARE et amélioré par SEDGENICK. Il consiste à segmenter le tableau initial en deux tableaux plus petits, puis de répéter le même processus sur chacun des tableaux ainsi obtenus jusqu'à ce que chaque nouveau tableau ne contient plus qu'un seul élément.

La méthode de segmentation d'un tableau donné consiste à sélectionner l'un de ses éléments comme séparateur. Tous les éléments inférieurs à cet élément seront alors placés dans le 1er sous tableau et tous les éléments supérieurs dans le second.

Plusieurs variantes de l'algorithme peuvent être obtenues selon la façon de choisir l'élément séparateur ou pivot. Par exemple :

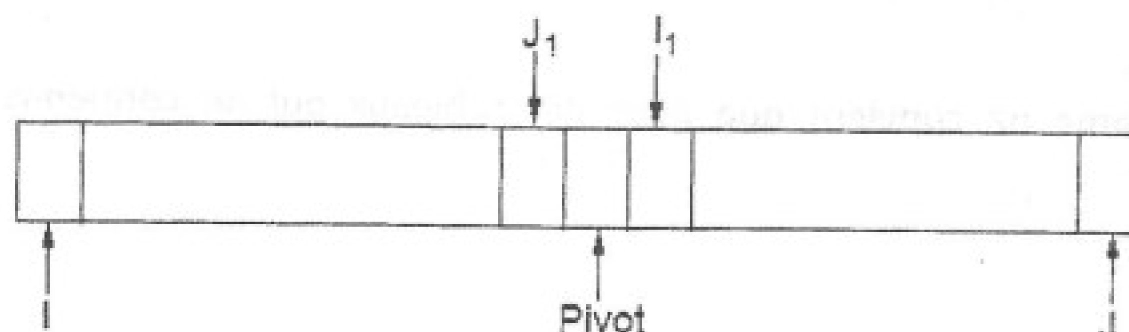
- Pivot= T[(Ent(N/2)] , élément milieu du tableau.
- Pivot égal à la valeur médiane des trois éléments de gauche, de droite et du milieu

La procédure de segmentation se compose de cinq étapes :

1- Choix du pivot
2- Balayage du tableau à segmenter du début vers la fin (ou de gauche à droite) jusqu'à trouver un indice I_1 tel que : $T[I_1]$ >= pivot.
3- Balayage du tableau dans le sens inverse jusqu'à trouver un indice J_1 tel que : $T[J_1]$ =< pivot
4- Si I_1 =< J_1 alors on effectue les opérations suivantes :
    - Echanger $T[I_1]$ avec $T[J_1]$
    - I_1 <-------I_1 + 1
    - J_1<-------J_1 + 1
5- On répète les étapes 2, 3 et 4 jusqu'à avoir $I_1 > J_1$

A l'issue de cette dernière étape le pivot est définitivement placé et le tableau est segmenté en deux parties :

- Une partie qui contient les éléments d'indice inférieur ou égal à J_1 et qui sont tous inférieurs au pivot
- Une partie qui contient les éléments d'indice supérieur ou égal à I_1 et qui sont tous supérieurs au pivot

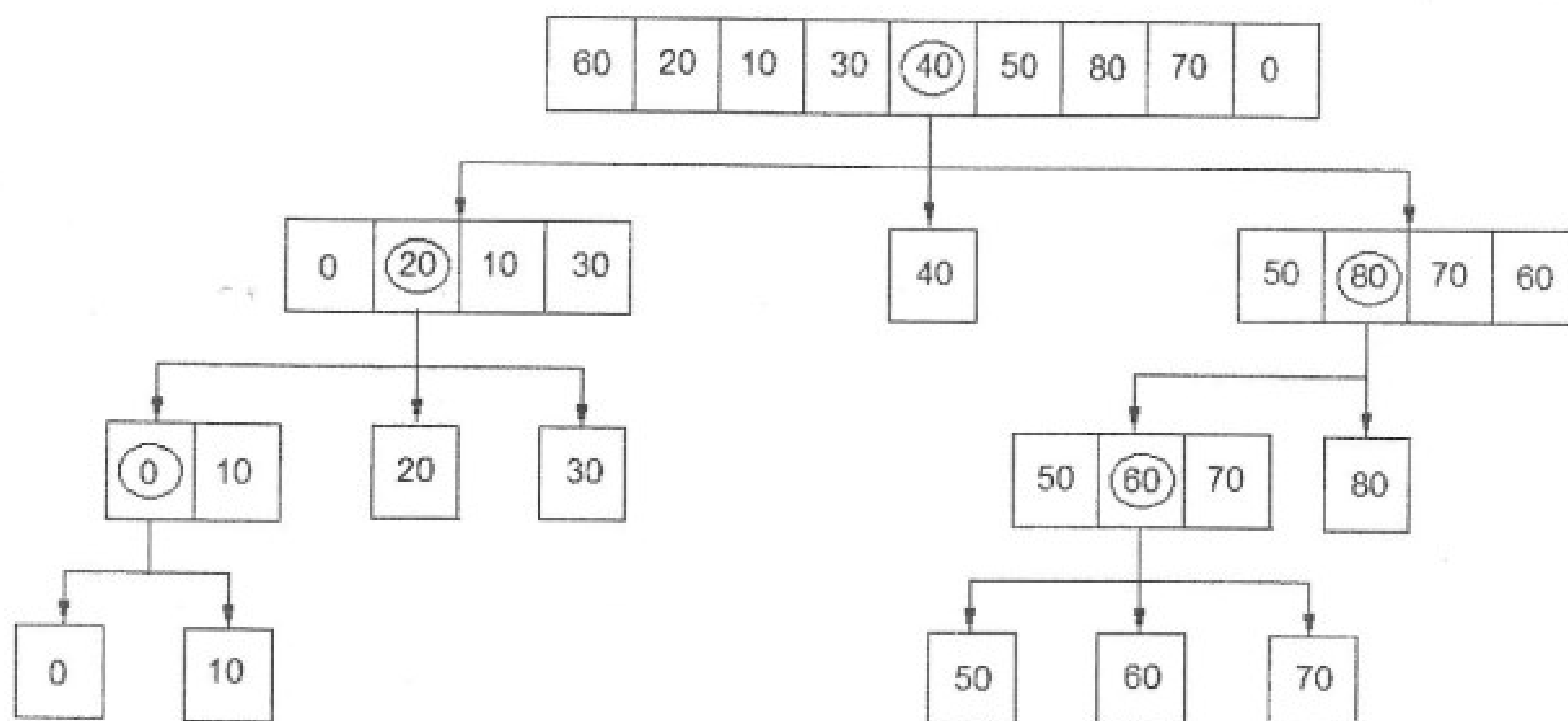


En conclusion, pour avoir le tableau final trié, il suffit de répéter récursivement la même procédure aux deux parties obtenues.

Les appels récursifs sont arrêtés lorsqu'on obtient des partitions à un seul élément. Dans ce cas on aura J_1=I et I_1=J .

Exemple :
Soit à trier le tableau suivant dans l'ordre croissant :

60	20	10	30	40	50	80	70	0



Algorithme :

Fonction Tri_rapide (T Tableau en Réel, I, J en Entier)
Début
Variable Pivot en Réel
Variables I_1, J_1 *en Entier*
I_1 <-------I
J_1<-------J
Pivot <------- T[Ent(I+J)/2)]
Tantque (I_1 =< J_1) *faire*
Début
Tantque (T[I_1] < Pivot) *faire*
Début
I_1 <------- I_1 + 1
Fin
Tantque (T[J_1] > Pivot) *faire*
Début
J_1 <-------J_1 - 1
Fin
Si I_1 =< J_1 *alors*
Echange(T[I_1], T[J_1])

```
            Finsi
            I1 <------- I1 + 1
            J1 <-------J1 - 1
        Fin
    Si I < J1 alors
            Tri_rapide(T, I, J1)
    Finsi
    Si I1 < J alors
            Tri_rapide(T, I1, J)
    Finsi
    Fin
```